Approximate Hybrid Binary-Unary Computing with Applications in BERT Language Model and Image Processing

Alireza Khataei
Department of ECE
University of Minnesota
Minneapolis, MN, USA
khata014@umn.edu

Gaurav Singh
Department of ECE
University of Minnesota
Minneapolis, MN, USA
singh431@umn.edu

Kia Bazargan
Department of ECE
University of Minnesota
Minneapolis, MN, USA
kia@umn.edu

ABSTRACT

We propose a novel method for approximate hardware implementation of univariate math functions with significantly fewer hardware resources compared to previous approaches. Examples of such functions include exp(x) and the activation function GELU(x), both used in transformer networks, gamma(x), which is used in image processing, and other functions such as tanh(x), cosh(x), sq(x), and sqrt(x). The method builds on previous works on hybrid binary-unary computing. The novelty in our approach is that we break a function into a number of sub-functions such that implementing each sub-function becomes cheap, and converting the output of the sub-functions to binary becomes almost trivial. Our method also uses self-similarity in functions to further reduce the cost. We compare our method to the conventional binary, previous stochastic computing, and hybrid binary-unary methods on several functions at 8-, 12-, and 16-bit resolutions. While preserving high accuracy, our method outperforms previous works in terms of hardware cost, e.g., tolerating less than 0.01 mean absolute error, our method reduces the (area × latency) cost on average by 5, 7, and 2 orders of magnitude, compared to the conventional binary, stochastic computing, and hybrid binary-unary methods, respectively. Ultimately, we demonstrate the potential benefits of our method for natural language processing and image processing applications. We deploy our method to implement major blocks in an encoding layer of BERT language model, and also the Roberts Cross edge detection algorithm. Both include non-linear functions.

CCS CONCEPTS

• **Hardware → Reconfigurable logic and FPGAs**; • **Computing methodologies** → *Neural networks*; *Image processing*.

KEYWORDS

hardware accelerators, approximate computing, unary computing, stochastic computing, BERT language model, image processing

ACM Reference Format:
Alireza Khataei, Gaurav Singh, and Kia Bazargan. 2023. Approximate Hybrid Binary-Unary Computing with Applications in BERT Language Model and Image Processing. In *Proceedings of the 2023 ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '23), February 12–14, 2023, Monterey, CA, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3543622.3573181

FPGA '23, February 12–14, 2023, Monterey, CA, USA

ACM ISBN 978-1-4503-9417-8/23/02…$15.00
https://doi.org/10.1145/3543622.3573181

1 INTRODUCTION

The past two decades have witnessed an evolution in computing from single-threaded, single-core processor architectures to ones with a diverse set of processing elements, with more specialized accelerators embedded in these platforms. Each application domain requires bespoke accelerators to implement its unique set of operators and tasks, optimized for the best performance. The explosion of machine learning applications has brought about highly optimized units such as matrix multiplication. Multiply-accumulate operations have been well-studied in the past decade, and FPGAs and other platforms usually provide enough computing resources for this common operation. What is missing, though, is efficient units performing non-linear operations.

A number of research groups have looked into non-traditional, non-binary number representations that enable low-cost implementation of non-linear functions. The next section will present the background on these methods, which include Stochastic Computing [27], Fully Unary [19, 20], and Hybrid Binary-Unary (HBU) Computing [7]. We take the exact HBU method and make structural modifications that significantly improve its LUT-count and critical path delay by allowing a controllable level of approximation. In contrast to the exact HBU method, our approach can implement functions at higher resolutions with ultra-low hardware costs.

Using FPGAs, we compare our method to the conventional binary, stochastic, and exact HBU methods on several functions at 8-, 12-, and 16-bit resolutions. We also apply our technique to the non-linear functions in the BERT language model and Roberts Cross edge detection algorithm to examine the impacts of approximation error on the quality of real systems. To explore the trade-off between accuracy and hardware costs, we synthesize two hardware designs with different error ranges for each function. The experimental results show that our method reduces the area × latency cost on average by 10^5, 10^7, and 10^2, compared to the conventional binary, stochastic computing, and exact HBU methods, respectively.

The key advantages of our proposed method over the exact HBU method are as follows:

(1) Our method can efficiently implement math functions at higher resolution (e.g., 16-bit) compared to the exact HBU method which is limited to 12-bit functions.
(2) Our method can provide a trade-off between accuracy and hardware cost, which is beneficial in applications that can

tolerate some approximation error. For instance, it can reduce hardware area by an average of 2 orders of magnitude by introducing less than 0.01 mean absolute error.

(3) Our method eliminates the need for a binary adder when assembling the results from sub-functions, which results in shorter latency and critical path delay.

2 BACKGROUND

2.1 Alternatives to Binary

For many years, the binary radix representation has been of great interest due to its compact size advantages in digital systems. An alternative to the conventional binary system is stochastic computing [10, 26–28, 33] which uses random streams of bits to represent numbers in the unit interval. To convert a w-bit binary number into this system, 2^w bits are needed, in which the ratio of 1's to the length of the stream determines the value as a probability. Using stochastic approaches, complex computations can be performed with a low-cost hardware architecture. Despite that, stochastic methods suffer from high latency due to the length of streams which is exponentially longer than binary[23]. Moreover, they do not provide accurate output results due to random variations[1]. The splitting resolution method [21] was proposed to reduce latency but at the expense of increasing area. In addition, new generations of stochastic computing ware proposed in [15, 22], in which computations are performed in a deterministic fashion but at the cost of increased latency [6].

Fully unary computing [19, 20] was introduced as an alternative, in which an arbitrary function can be implemented using a network of wires and XOR gates called the "scaling network". This method converts binary input numbers to unary representation as thermometer codes using an "encoder", e.g., the binary number 101_2 is converted to 1111100, which is a code of length 7 (the maximum value that a 3-bit binary number can hold), with the first five bits set to 1. After the conversion, it performs computations on the unary value using the scaling network, and finally converts the unary result to binary output using a "decoder". As the authors in [19] report, the fully unary method outperforms the conventional binary and stochastic computing methods such as Bernstein polynomial[29], Maclaurin series [30], linear finite-state machines[17], and dynamical systems with feedback[35] methods. However, as the resolution increases beyond 12-bit, the complexity of decoders and encoders increases exponentially which makes the overall architecture unattractive in terms of area × latency. To address this issue, exact hybrid binary-unary (exact HBU) was proposed by [6] which takes advantage of unary and binary at the same time. The lower bits of input data are converted from binary to unary while keeping the upper bits in the binary format. Recently, many researchers have been exploring the unary methods and deploying them to accelerate compute-intensive applications ranging from conventional neural networks [5, 8], to image processing, and sorting networks[23].

2.2 Unary Number Representation

In general, a unary number format is any representation in which all bits have the same weight, as opposed to the binary radix that assigns a specific weight to each bit's position[25]. Fully unary computing [19], exact HBU [6] use a unary representation in the form of thermometer codes: a w-bit binary number gets expanded to 2^w bits, in which the total number of 1's in those 2^w bits determines the value carried by the bundle of wires. For instance:

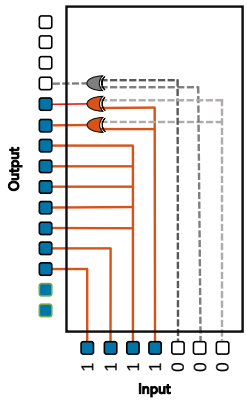
$$(000)_{\text{Unary}} = (00)_{\text{Binary}}, (100)_{\text{Unary}} = (01)_{\text{Binary}}$$
$$(110)_{\text{Unary}} = (10)_{\text{Binary}}, (111)_{\text{Unary}} = (11)_{\text{Binary}}$$

Since there is a one-to-one correspondence between these two representations, unary thermometer codes can express many types of numbers, including integer, fixed-point, signed, and unsigned, which are basically based on our interpretation of their corresponding binary formats. As an example, $(111)_{\text{Unary}}$ is equal to $(11)_{\text{Binary}}$, and $(11)_{\text{Binary}}$ may be interpreted as $(3)_{\text{Decimal}}$ or $(1.5)_{\text{Decimal}}$ or $(-1)_{\text{Decimal}}$ or other numbers.

2.3 Fully Unary Computing [19, 20]

Generally speaking, a complex math function in digital hardware can be considered as a black box following a look-up table that maps an N-bit input to an M-bit output.

In fully unary computing, a binary input is encoded to the unary thermometer code feeding a network of wires and XOR gates, in which the unary input is mapped to output wires that form a new thermometer code equal to the desired value. At the final stage, the output is decoded from unary to binary. Fig. 1 (left) shows the structure of the scaling network corresponding to the function described as a look-up table in Fig. 1 (right). The encoder and decoder units to convert binary-to-unary as input, and convert the unary output back to binary are not shown in the figure.



Input	Output
0000000	110000000000000
1000000	111000000000000
1100000	111100000000000
1110000	111111110000000
1111000	111111111100000
1111100	111111111110000
1111110	111111111100000
1111111	111111110000000

Figure 1: The structure of the scaling network and its corresponding look-up-table in the unary domain

As shown in Fig. 1, if the input increases from x = 2 to x = 3, the output goes up by 5 steps from f = 4 to f = 9. Therefore, X_3 (the 3^{rd} bit of the unary input) is connected to F_5 (the 5^{th} bit of the unary output), F_6, F_7, F_8, and F_9 using wires. If the function was monotonically increasing, it could be implemented entirely out of such scaling wires. The arbitrary function in Fig. 1 is, however, non-monotonic, and if the input increases from x = 5 to x = 6, the output goes 1 step down from f = 12 to f = 11. As a result, we must use an XOR gate to flip F_{12} from '1' to '0'. More precisely, F_{12} first changes from '0' to '1' when the input reaches (or exceeds) x = 5, and it later switches back to '0' when the input reaches (or exceeds)

x = 6. Therefore, X_5 and X_6 are triggering points for F_{12} and must be connected to it through an XOR gate, i.e., $F_{12} = X_5 \oplus X_6$

2.4 Exact HBU Computing [6]

For high-resolution and non-monotonic functions, fully unary computing is not competitive with its counterparts due to the complexity of thermometer encoders and decoders. To address this issue, exact HBU[6] can be deployed to reduce the complexity of the decoders and encoders by dividing the original function into n sub-functions with their offsets separated. Since the input and output lengths of each sub-function are shortened, they can be implemented using the fully unary cores at lower resolutions. However, since all the sub-functions are implemented independently and fed concurrently, we have n different outputs computed by fully unary cores, one of which contributes to the final expected value. The upper bits of the binary input determine the correct output and its corresponding offset that was separated before. Eventually, the correct output is added with its offset to generate the final binary output. Fig. 2 shows the overall architecture of the exact HBU method.

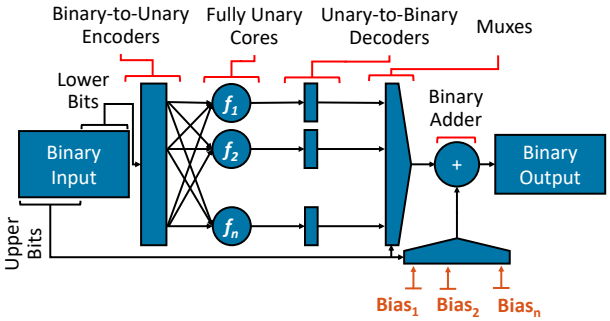


Figure 2: The structure of the exact HBU method [6].

3 PROPOSED WORK

We will first provide an overview of our method (Sec. 3.1), followed by the pseudo-code of our method (Sec. 3.2), followed by a small numeric example showing how we implement the functions (Sec. 3.4).

3.1 Overview

As discussed earlier, the exact HBU method[6] divides a math function $f(x)$ into n sub-functions and separate their offsets:

$$f(x) = \begin{cases} b_1 + f_1(x) & 0 \le x < x_1 \\ \dots & \\ b_n + f_n(x) & x_{n-1} \le x < x_n \end{cases} \quad (1)$$

where f_i, b_i, and n represent the i^{th} sub-function without offset, offset, and the number of divisions, respectively.

In our method, we propose an algorithm that divides and transforms the original function $f(x)$ into $\hat{f}(x)$ in such a way that the upper K bits of the output in each sub-function are the same in the binary domain.

$$f(x) \longrightarrow \hat{f}(x) = \begin{cases} \hat{f}_1(x) & 0 \le x < x_1 \\ \dots & \\ \hat{f}_n(x) & x_{n-1} \le x < x_n \end{cases} \quad (2)$$

Since the upper K bits of each sub-function $\hat{f}_i(x)$ are fixed, we can separate those K bits and rewrite $\hat{f}(x)$ as follows using simple bit-concatenation:

$$f(x) \longrightarrow \hat{f}(x) = \begin{cases} \{"UB_1", g_1(x)\} & 0 \le x < x_1 \\ \dots & \\ \{"UB_n", g_n(x)\} & x_{n-1} \le x < x_n \end{cases} \quad (3)$$

where "UB_i" represents the upper K bits of the i^{th} sub-function, and $g_i(x)$ represents the i^{th} truncated sub-function which is derived from $\hat{f}_i(x)$ by omitting its upper K bits. Therefore, the concatenation of "UB_i" and $g_i(x)$ creates $\hat{f}_i(x)$.

Since we have divided the input length of the function into n sub-regions, we need to use n fully unary cores to implement $g_i(x)$, $i \in \{1, 2, ..., n\}$. Similar to the exact HBU method, n different outputs are computed in parallel, one of which must contribute to the final binary output. Therefore, we need to use the upper bits of the input to multiplex the correct output. Fig. 3 shows the overall architecture of our method.

Using such a transformation, each approximated sub-function $g_i(x)$ can be implemented in the unary domain at a lower resolution than the original function $f(x)$. It means we can utilize low-cost thermometer encoders and decoders to perform the computations, which are the primary bottlenecks of the previous fully unary and exact HBU methods. In addition, unlike the exact HBU, our method does not use any offsets b_i, hence eliminating the need for any binary adders, which translates to much lower latency and critical path delay.

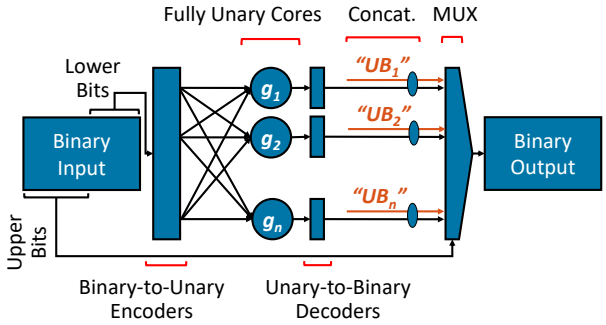


Figure 3: The structure of our method.

This transformation can beat the previous exact HBU method in terms of hardware area, critical path delay, and latency. In addition, we propose a self-similarity measurement technique to further reduce hardware costs. Based on our observation, many of the transformed sub-functions are either the same or very similar. Therefore, using some level of approximation, we can dramatically reduce the number of distinct fully unary cores and decoders that need to be implemented. This also enables us to implement complex math functions at higher resolutions and lower costs, which was not feasible in the previous exact HBU approach. These optimization strategies make our method superior to conventional binary, stochastic, and exact HBU methods in terms of area, critical path delay, and latency, at 8- to 16-bit resolutions. In the following sections, we will discuss our method in detail.

Table 1: The parameters used in our proposed algorithm.

Parameter	Description
IL	Initial division length
L_{min}	Sub-function minimum input length
K	# Upper bits to be fixed in sub-functions
TRE	Target rounding error
TSE	Target similarity error

3.2 Function Division and Transformation

In this section, we focus on how to efficiently divide and transform an arbitrary function to reduce hardware cost and approximation error. First, we define 5 parameters that are used in our algorithm, two of which (IL and $Lmin$) were inspired by the exact HBU method[6]. Table 1 describes the parameters and their definitions.

Algorithm 1 shows the function division and transformation process for an arbitrary math function $f(x)$ at the input and output resolutions w_{in} and w_{out}. In this process, the input and output values of the functions are considered an unsigned integers, although they may originally represent other values. As a result of this algorithm, g(x), UB, and $gBoundaries$ are returned as outputs, where $gBoundaries$ is a list of tuples that determines the input range of each $g_i(x)$, and UB is the list of "UB_i"s.

In the first loop (lines 6–9), the input domain $[0, 2^{w_{out}} - 1]$ is divided into IL-bit sub-regions, and the results are stored in a list called $fBoundaries$. In fact, $fBouandaries$ is a list of tuples, each of which corresponds to the input domain of a sub-function that still requires transformation and processing. Once a sub-region is processed and finalized, it is stored in $gBouandaries$. In the next loop (lines 10–39), in each iteration, we select an input sub-region from $fBouandaries$ and remove that from the list. Next, we check whether the upper K bits of $f(x)$ in this region are fixed or not. If they are fixed, we store this sub-region into $gBouandaries$ and the corresponding output values $f(x)$ into $\hat{f}(x)$. If they are not fixed, we round the sub-function output to the nearest values whose upper K bits are fixed in this sub-region. Among all the 2^K options for the upper bits' format, we choose the one which results in the minimum rounding error. Next, if this minimum rounding error is less than TRE or the length of the sub-region is equal to L_{min}, we store this sub-region into $gBouandries$ and the corresponding rounded values f(x) into $\hat{f}(x)$. Otherwise, we divide this sub-region by 2 and store them into $fBouandries$. The rounding error is calculated based on the mean absolute error equation as follows:

$$RoundingError = \frac{1}{m}\sum_{x}\left|\frac{f_{temp}(x) - f_{sub}(x)}{2^{w_{out}}}\right| \tag{4}$$

At the end of the while loop, $gBouandries$ contains a list of tuples that corresponds to the input domains of sub-functions whose upper K bits are fixed. Therefore, we separate these upper bits from $\hat{f}(x)$ and store the resulting values as $g(x)$. The upper bits of each sub-function are also stored in a list called UB.

Using the algorithm above, an arbitrary function $f(x)$ can be transformed and divided into sub-functions $\hat{f}_i(x)$ whose upper K bits are fixed. Thus, we can temporarily eliminate the upper bits, and implement each truncated sub-function (denoted as $g_i(x)$) in the fully unary domain using only the lower bits, which in turn reduces the complexity of decoders and encoders. However, once the unary computations are performed and converted to the binary format, the output of each $g_i(x)$ must be concatenated with their

Algorithm 1: Function Division and Transformation

1 **Parameters:** IL, L_{min}, K, TRE
2 **Input:** $f(x)$
3 **Output:** $g(x), UB, gBoundaries$
4 $x_s \leftarrow 0$;
5 // (6-9) Dividing the input region by 2^{IL}
6 **for** $i = 1$ to $2^{w_{in}-IL}$ **do**
7 $\quad fBoundaries$.push($[x_s, x_s + 2^{IL} - 1]$)
8 $\quad x_s \leftarrow x_s + 2^{IL}$
9 **end**
10 **while** $fBoundaries \neq NULL$ **do**
11 $\quad [x_s, x_e] \leftarrow fBoundaries.pop()$
12 $\quad f_{sub} \leftarrow f([x_s, x_e])$
13 $\quad$ // (14-15) Finding min / max of K upper bits
14 $\quad minUB \leftarrow \lfloor min(f_{sub})/2^{w_{out}-K} \rfloor$
15 $\quad maxUB \leftarrow \lfloor max(f_{sub})/2^{w_{out}-K} \rfloor$
16 $\quad$ **if** $minUB = maxUB$ **then**
17 $\quad\quad gBoundaries$.push($[x_s, x_e]$)
18 $\quad\quad \hat{f} \leftarrow f_{sub}$
19 $\quad$ **else**
20 $\quad\quad$ // (21-29) Computing the lowest $RoundingError$ for the K upper bits
21 $\quad\quad BestRE \leftarrow inf$
22 $\quad\quad$ **for** $i = 0$ to $2^K - 1$ **do**
23 $\quad\quad\quad f_{temp} \leftarrow (i \times 2^{wout-K}) + (f_{sub} \mod 2^{w_{out}-K})$
24 $\quad\quad\quad RE \leftarrow RoundingError(f_{sub}, f_{temp})$
25 $\quad\quad\quad$ **if** $RE < bestRE$ **then**
26 $\quad\quad\quad\quad bestRE \leftarrow RE$
27 $\quad\quad\quad\quad bestUB \leftarrow i$
28 $\quad\quad\quad$ **end**
29 $\quad\quad$ **end**
30 $\quad\quad$ // (31-37) Applying the approximation or dividing the sub-region by 2
31 $\quad\quad$ **if** $bestRE < TRE$ or $L_{min} = (x_e - x_s)$ **then**
32 $\quad\quad\quad gBoundaries$.push($[x_s, x_e]$)
33 $\quad\quad\quad \hat{f} \leftarrow bestUB \times 2^{w_{out}-K} + f_{sub} \mod 2^{w_{out}-K}$
34 $\quad\quad$ **else**
35 $\quad\quad\quad fBoundaries$.push($[x_s, (x_e - 1)/2]$)
36 $\quad\quad\quad fBoundaries$.push($[(x_e + 1)/2, x_e]$)
37 $\quad\quad$ **end**
38 $\quad$ **end**
39 **end**
40 $g \leftarrow \hat{f} \mod 2^{w_{out}-K}$
41 $UB \leftarrow \lfloor \hat{f}/2^{w_{out}-K} \rfloor$

separated upper bits. At the final stage, the upper bits of the binary input are used to multiplex the expected final values among the results provided by each unary core.

3.3 Self-Similarity Measurement

In the previous subsection, we discussed how to divide and transform an arbitrary function $f(x)$ into n sub-functions $\hat{f}_i(x)$ whose upper K bits are fixed. Therefore, each $\hat{f}_i(x)$ can be rewritten as the concatenation of "UB_i" and $g_i(x)$. As a result, we need n separate fully unary cores to correspond to each $g_i(x)$ to perform the computation. Using the aforementioned method alone can reduce the hardware area and latency significantly. However, we can further reduce the hardware utilization by implementing similar unary cores as one unit. Hence, we need to measure the self-similarity in all the truncated sub-functions $g_i(x)$ which have the same input lengths.

To do so, a new parameter TSE is defined, and each $g_i(x)$ is compared with all the other $g_j(x)$'s. The similarity error between $g_i(x)$ and $g_j(x)$ can be computed based on the following equation:

$$SimilarityError = \frac{1}{m}\sum_{x}\left|\frac{g_i(x) - g_j(x)}{2^{w_{out}}}\right| \tag{5}$$

If the similarity error between $g_i(x)$ and $g_j(x)$ is less than or equal to TSE, they can be implemented interchangeably. In other words:

$$\text{If } \exists(g_i, g_j) \text{ such that } SimilarityError(g_i, g_j) \le TSE$$
$$\Rightarrow g_i(x) \approx g_j(x)$$

3.4 Guiding Example

To clarify the proposed method, we go through the steps using a guiding example. Fig. 4 (top-left) shows an arbitrary function $f(x)$ with $w_{in} = 4$ bits and $w_{out} = 4$ bits. In this example, we assume $IL = 3$ bits, $L_{min} = 2$ bits, $K = 2$ bits, $TRE = 2.5 \times 10^{-2}$, and $TSE = 0.5 \times 10^{-2}$. As $w_{in} = 4$, and $IL = 3$, we initially divide the input domain ([0 15]) into 2 equal-sized sub-regions ([0, 7], and [8, 15]) and store them in the list $fBoundaries$. In the first iteration of the while loop, we select the first item from $fBoundaries$ (i.e., [0, 7]) and remove it from the list. In this sub-region, the output values $f_{sub} = f([0, 7])$ ranges from 0 to 3, hence the upper $K = 2$ bits are fixed. As a result, we store this sub-region into $gBoundaries$ and f_{sub} values into $\hat{f}$. In the next iteration, we select the next item from $fBoundaries$ ([8, 15]) and remove it from the list. In this sub-region, $f_{sub} = f([0, 7]) = \{4, 4, 4, 7, 8, 11, 12, 8\}$, therefore, the upper 2 bits are not fixed. We have $2^K = 4$ options for the upper bits' format, which are '00', '01', '10', and '11'. We consider all the options separately and round the values to the nearest values whose upper 2 bits are the same. In what follows, we illustrate the possible options for rounding the upper bits:

(1) **'00XX'**$(i = 0)$**:** $f_{temp} = \{3, 3, 3, 3, 3, 3, 3, 3\}$
(2) '01XX'$(i = 1)$**:** $f_{temp} = \{4, 4, 4, 7, 7, 7, 7, 7\}$
(3) '10XX'$(i = 2)$**:** $f_{temp} = \{8, 8, 8, 8, 8, 11, 11, 8\}$
(4) '11XX'$(i = 2)$**:** $f_{temp} = \{12, 12, 12, 12, 12, 12, 12, 12\}$

Among all these options, '01XX' results in the best (lowest) rounding error, which is 8.59×10^{-2}. However, since the error is not less than TRE ($8.59 \times 10^{-2} > 2.5 \times 10^{-2}$), and the sub-region length has not reached the minimum allowable length L_{min} ($3 \neq 2$), we do not apply the approximation for this sub-region. Instead, we divide the sub-region by 2 ([8, 11] and [12, 15]) and add them into $fBoundaries$. At this time, $fBoundaries$ still has 2 elements, and we must go through the while loop again. In the next iteration, $f_{sub} = f([8, 11])$ ranges from 4 to 7, which means the upper 2 bits are fixed. Hence, we add the input sub-region to $gBoundaries$, which is a list of sub-regions that are finalized, and we add the transformed output of this sub-region in $\hat{f}$. In the next iteration, $f_{sub} = f([12, 15])$ ranges from 8 to 12, which means the upper 2 bits are not fixed. We again consider all 4 possible options and pick the best one which minimizes the rounding error. The best option that we have is to fix the upper bits as '10' which results in the rounding error 1.56×10^{-2}. Since the error is less than TRE, we keep the changes and add this sub-function into $\hat{f}$ and $gBoundaries$.

At this point, there is no region left in the queue ($fBoundaries$), and we get out of the while loop. At the last stage, we store the upper K bits and the remaining lower bits of $\hat{f}$ into g and UB, respectively. As a result of this algorithm, the original function was divided into 3 different sub-functions in which the upper 2 bits are fixed, and they can be safely truncated before performing the computations in the unary domain. Fig. 4 shows the truncated sub-functions (bottom middle) and their corresponding upper bits (bottom right) as well as the intermediate steps of the algorithm.

Since the original function was divided into 3 sub-regions corresponding to $x \in [0, 7]$, $[8, 11]$, and $[12, 15]$, we need 3-to-7, 2-to-3 and 2-to-3 thermometer encoders (determined by the x range of sub-functions), followed by 3 fully unary cores which implement the truncated sub-functions. These encoders, use the lower bits of binary input to feed the unary cores. As the output of each truncated sub-function ranges from 0 to 3, each core is connected to a 3-to-2 thermometer decoder separately. Next, we concatenate each output with its corresponding eliminated upper bits. Finally, the upper bits of the binary input are used to multiplex the correct output result for that particular input.

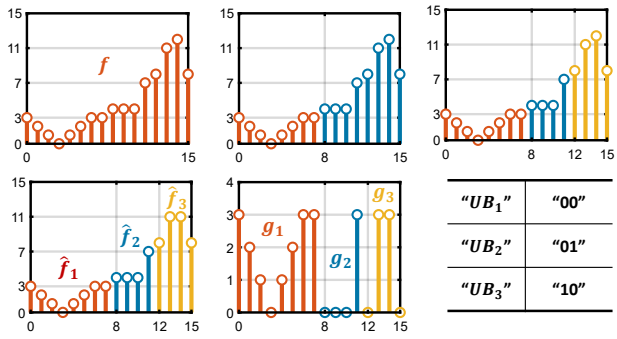


Figure 4: Function division and transformation of an arbitrary function using our proposed algorithm.

To further reduce the hardware cost, we can check the possibility of implementing similar truncated sub-functions as one unary core. Thus, we compare all the truncated sub-functions with the same input lengths together. In our example, there are two regions with the same input length which correspond to $x \in [8, 11]$, and $x \in [12, 15]$. The similarity error between these two is equal to 0.14 which is greater than TSE, and hence these two regions are not similar enough to be implemented interchangeably.

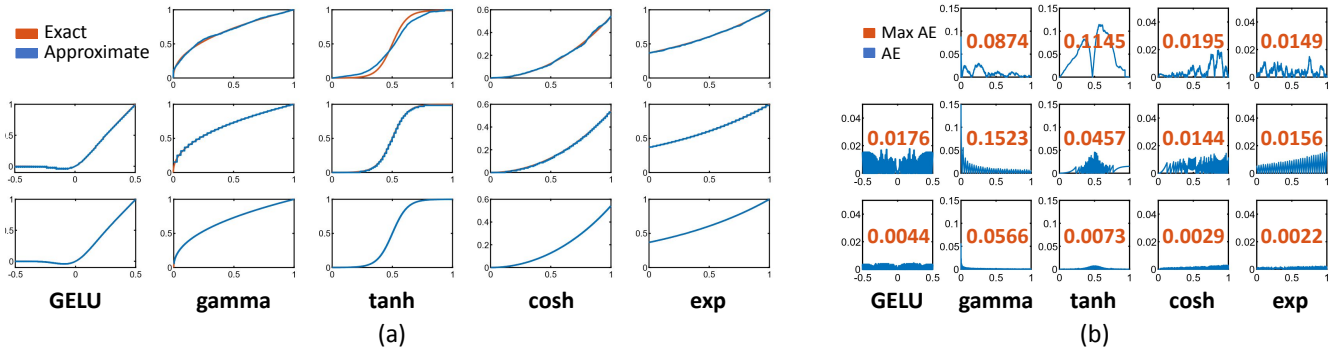


Figure 5: (a) Graph, (b) absolute error (AE) and maximum absolute error (Max AE) of each function at 12-bit resolution using conventional stochastic computing method [27] (top row), our method with *Config. 1* (middle row) and *Config. 2* (bottom row).

4 EVALUATION

In this section, we evaluate our proposed method in terms of hardware implementation and accuracy against the conventional binary, stochastic[27] and exact HBU[6] methods on a number of functions listed in Table 2. FloPoCo[2] is another method that implements non-linear functions using polynomial approximations. As part of its architecture, it uses multipliers to perform the computations and if we force the synthesizer to implement them using LUTs (instead of DSP blocks), the architecture would become inefficient in terms of area. On the other hand, this method implements the functions more accurately compared to our method. As a result, we did not compare our method against FloPoCo, as that would be comparing apples to oranges.

Finally, we apply our method to the Roberts Cross edge detection algorithm as an image processing application, and to the non-linear functions of BERT, to examine the effects of the approximations on a real system's quality and performance.

Table 2: Implemented math functions

Function Name	Equation
GELU	$0.5x(1+\frac{2}{\sqrt{\pi}}\int_0^{\frac{x}{\sqrt{2}}} e^{-t^2}dt)$
gamma	$x^{0.45}$
tanh	$(1+tanh(4(2x-1)))/2$
cosh	$cosh(x)-1$
exp	e^{x-1}

4.1 Efficiency vs Accuracy Trade-Off

By changing the parameters of the proposed algorithm, we can generate architectures with different hardware costs and accuracy. In practice, the amount of tolerable error varies from one application to another. Since our method is a trade-off between hardware costs and accuracy, we synthesize two hardware architectures per function which are denoted as *Configuration 1*, and *Configuration 2*. These refer to those architectures introducing less than 0.01 and 0.001 mean absolute error, respectively. However, we will show that both configurations in our method cause on average less error compared to the stochastic approach, besides requiring fewer hardware resources.

4.2 Hardware Implementation

To evaluate our method in terms of hardware costs and accuracy, we used Xilinx's XC7K70TFBG676-2 FPGA and Vivado 2020.2 to implement the math functions listed in Table 2 using different methods including the conventional binary, stochastic, exact HBU, and our proposed method with two configurations. To generate hardware designs, we developed a Matlab program to generate Verilog files. It changes the parameters of the proposed algorithm in predefined ranges to find the optimum hardware design for implementing an arbitrary function with desired accuracy and resolution. We implemented the functions at 8-, 12-, and 16-bit resolutions with no DSP48 and BRAM blocks. Table 3 shows the FPGA hardware costs and accuracy results. "Tcrit" and "Cy" denote the critical path delay in nanoseconds and the number of clock cycles needed to execute, respectively, and "A×L" denotes area × latency cost, which is the multiplication of the Area, T_{crit}, and Cy columns. The Error column shows the total approximation error based on the mean absolute error metric. Fig. 5 (a) shows the graph of each function implemented using conventional stochastic computing and our proposed methods. Fig. 5 (b) compares the accuracy and maximum absolute error of the approximated functions implemented on the FPGA. Except for the *GELU* function, to be consistent with the stochastic method, the input and output data were scaled to fit the unit interval [0, 1], i.e., the binary input and output were interpreted as w-bit fixed-points with w-bit fractions.

In the conventional binary implementations, we used fully pipelined Xilinx CORDIC (v6.0) and Divider Generator (v5.1) LogiCOREs to perform the CORDIC-based functions such as $tanh(x)$, $cosh(x)$, e^x. To implement the function *gamma* in binary, we approximated it using a Taylor polynomial of degree 5 around $x = 0.5$ and converted the polynomial's coefficients into fixed-point formats. Finally, to implement the *GELU* activation function in binary, we approximated the equation of Table 2 with:

$$GELU \approx 0.5x(1+tanh[\sqrt{\tfrac{2}{\pi}}(x+0.044715x^3)] \quad (6)$$

and implemented it using adders and multipliers.

Table 3: FPGA hardware costs and accuracy results.

Func	Area (LUT)	T_{crit} (ns)	Cy	A×L	Error	Area (LUT)	T_{crit} (ns)	Cy	A×L	Error	Area (LUT)	T_{crit} (ns)	Cy	A×L	Error
	Conv. Binary Method with W = 8					Conv. Binary Method with W = 12					Conv. Binary Method with W = 16				
GELU	1415	16.09	21	478114.35	3.03E-03	2170	19.59	29	1232798.7	1.95E-04	3190	22.38	38	2712904	1.20E-05
gamma	3024	24.74	1	74813.76	9.31E-03	4341	29.07	1	126192.9	6.16E-03	5299	33.14	1	175608.9	6.10E-03
tanh	440	1.80	21	16632	-	841	1.87	29	45607.43	-	1490	1.93	38	109276.6	-
cosh	332	1.80	11	6573.6	-	643	1.87	15	18036.15	-	1159	1.93	20	44737.4	-
exp	353	1.80	11	6989.4	-	668	1.87	15	18737.4	-	1197	1.93	20	46204.2	-
Avg.	**1112.8**	**9.25**		**116624.62**		**1732.6**	**10.85**		**288274.51**		**2467**	**12.26**		**617746.1**	
	Conv. Stochastic Method [27] with W = 8					Conv. Stochastic Method [27] with W = 12					Conv. Stochastic Method [27] with W = 16				
gamma	105	2.88	256	77414.4	1.13E-02	108	3.27	4096	1446543	1.00E-02	147	3.62	65536	34874327	1.07E-02
tanh	145	3.80	256	141056	4.03E-02	180	3.88	4096	2860646	4.71E-02	236	4.1	65536	63412634	3.47E-02
cosh	103	2.86	256	75412.48	7.88E-02	107	3.37	4096	1476977	4.45E-03	143	3.31	65536	31020155	1.70E-03
exp	99	3.4	256	86169.6	1.18E-02	108	3.26	4096	1442120	4.02E-03	153	4.09	65536	41010463	2.35E-03
Avg.	**113**	**3.24**		**95013.12**		**125.75**	**3.45**		**1806572**		**169.75**	**3.78**		**42579395**	
	Exact HBU Method [6] with W = 8					Exact HBU Method [6] with W = 12					Exact HBU Method [6] with W = 16				
GELU	19	1.39	1	26.41	-	337	2.47	1	832.39	-	[Too Long to Synthesize]				
gamma	28	1.53	1	42.84	-	491	2.4	1	1178.4	-					
tanh	24	1.39	1	33.36	-	382	2.35	1	897.7	-					
cosh	26	1.39	1	36.14	-	430	2.43	1	1044.9	-					
exp	28	1.29	1	36.12	-	464	2.44	1	1132.16	-					
Avg.	**25**	**1.4**		**34.97**		**420.8**	**2.42**		**1017.11**						
	Our Method with W = 8 (*Config. 1*)					Our Method with W = 12 (*Config. 1*)					Our Method with W = 16 (*Config. 1*)				
GELU	3	0.95	1	2.85	4.96E-03	11	1.04	1	11.44	2.76E-03	13	1.35	1	17.55	2.12E-03
gamma	3	0.96	1	2.88	5.95E-03	7	1.14	1	7.98	6.64E-03	9	1.33	1	11.97	7.35E-03
tanh	4	1.29	1	5.16	9.32E-03	4	1.05	1	4.2	9.10E-03	13	1.55	1	20.15	7.79E-03
cosh	3	0.95	1	2.85	9.99E-03	6	1.05	1	6.3	4.34E-03	14	1.37	1	19.18	7.38E-03
exp	4	0.95	1	3.8	7.07E-03	9	1.14	1	10.26	4.60E-03	12	1.67	1	20.04	4.92E-03
Avg.	**3.4**	**1.02**		**3.51**		**7.4**	**1.08**		**8.04**		**12.2**	**1.45**		**17.78**	
	Our Method with W = 8 (*Config. 2*)					Our Method with W = 12 (*Config. 2*)					Our Method with W = 16 (*Config. 2*)				
GELU	13	1.22	1	15.86	7.17E-04	29	1.53	1	44.37	6.78E-04	46	1.41	1	64.86	5.23E-04
gamma	17	1.23	1	20.91	8.39E-04	38	1.25	1	47.5	7.45E-04	50	1.25	1	62.5	9.05E-04
tanh	16	1.23	1	19.68	9.77E-04	39	1.38	1	53.82	9.93E-04	42	1.4	1	58.8	9.88E-04
cosh	14	1.22	1	17.08	5.19E-04	36	1.54	1	55.44	6.67E-04	36	1.54	1	55.44	7.23E-04
exp	14	1.22	1	17.08	5.80E-04	35	1.54	1	53.9	6.48E-04	55	1.53	1	84.15	5.96E-04
Avg.	**14.8**	**1.22**		**18.12**		**35.4**	**1.45**		**51**		**45.8**	**1.43**		**65.15**	

As a result of using these approximations and fixed-point representation, the functions *gamma* and *GELU* introduce errors in our binary implementations. However, we could have further reduced those errors, but at the expense of increasing hardware costs. As the experimental results show, the area × latency cost of our method with *Config. 1* is on average 3×10^{-5}, 2.79×10^{-5}, and 2.87×10^{-5} times the conventional binary method at 8-, 12-, and 16-bit resolutions. For the functions implemented with *Config. 2*, which preserves more accuracy than *Config. 1*, these ratios change to 1.55×10^{-4}, 1.77×10^{-4}, and 1.05×10^{-4}, respectively.

For the stochastic computing method, we used the SCSynth tool[9] to generate Verilog files describing the stochastic computing architectures. We did not implement the *GELU* activation function using the stochastic method since it does not satisfy a necessary condition of the Bernstein polynomial approximation, i.e., for some $x \in (0, 1)$, $f(x) = 0$ [29]. On average, the area × latency cost of our method with *Config. 1* is 3.69×10^{-5}, 4.45×10^{-6}, and 4.18×10^{-7} times the conventional stochastic computing method at 8-, 12-, and 16-bit resolutions, respectively. For *Config. 2*, the average area × latency cost turns into 1.91×10^{-4}, 2.82×10^{-5}, and 1.53×10^{-6} times the conventional stochastic computing method at 8-, 12-, and 16-bit resolutions, respectively. Besides these improvements, both configurations of our method cause less error than the stochastic computing approach, on average.

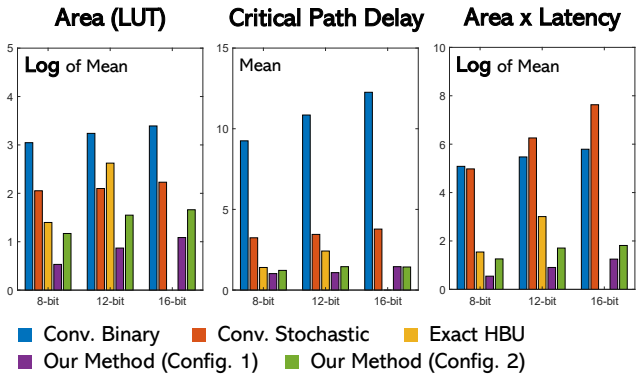


Figure 6: Comparison of hardware costs results. The scales of area and area × latency values are logarithmic.

For the exact HBU implementation, the exact HBU method took a long time and failed to synthesize the functions at 16-bit resolution in a reasonable amount of time due to the massive number of sub-functions and complex unary cores generated by this method. Therefore, we just presented the results at 8- and 12-bit resolutions. On average, the area × latency cost of our method with *Config. 1* is 10.04% and 0.79% of the exact HBU method at 8- and 12-bit resolutions, respectively. Using *Config. 2*, the cost of our method becomes 51.81% and 5.01% of the exact HBU at 8- and 12-bit resolutions, respectively.

Fig. 6 compares the hardware costs of implementing the functions using our method against the conventional binary, stochastic, and exact HBU (the scales of Area and Area × Latency values are logarithmic). The comparison shows that our method outperforms the other methods in terms of hardware cost. As we can see, the gap between the hardware cost of our method and other methods widens at higher resolutions.

4.3 Application: BERT Language Model

Transformer network was proposed in [31] as a deep learning model and an alternative to convolutional and recurrent neural networks. It uses the self-attention mechanism to process the sequential inputs in many natural language processing (NLP) tasks [11, 12]. Since Transformer can be relatively easily parallelized, using it in NLP tasks allows for fast training even when high-accuracy results are sought [31]. Transformer-based language models like BERT [3] have delivered high-quality results for various NLP downstream tasks, and RoBERTa [18] extended the work done in BERT and achieved state-of-the-art results in various benchmarks like GLUE [34].

While these models used floating-point, later works have quantized transformer based networks. I-BERT [16] specifically quantized RoBERTa while using purely integer operations. Their work removed the need to convert the output from 8-bit integer (INT8) based matrix multiplication into floating-point to apply non-linear operations like Softmax and GELU. Hence, they achieved an end-to-end integer based model. Their work used a polynomial approximation on the exponential function (for the Softmax) and the erf function (in GELU). The primary limitations of their work were in using 32-bit based INT division and multiplication operations in addition to a larger error rate in GELU relative to the bit-length of their operation. I-BERT also needs to store a significant number of scaling factors and coefficients to perform non-linear operations.

In this part, we employ our proposed method to implement non-linear functions very efficiently, hence providing better opportunities for higher levels of parallelism given the same area budget as existing implementations in vector processing. The end result would be an increase in overall system performance. Fig. 7 shows the overall architecture of a BERT encoder. Our innovation is to break non-linear functions into smaller sub-functions and use our method only for the highly non-linear parts of these functions at lower resolutions to reduce the hardware cost and accelerate the computations.

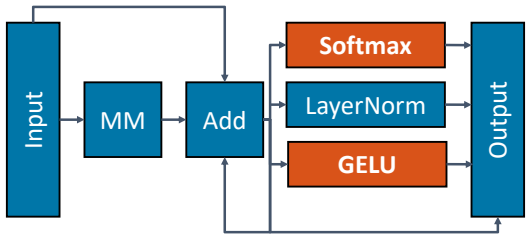


Figure 7: The architecture for the BERT Implementation.

Softmax is an activation function that transforms an input vector of N elements into a vector of size N representing a probability distribution. This function can be defined as:

$$Softmax(X)_i = \frac{e^{x_i}}{\sum_{j=1}^{N} e^{x_j}} \tag{7}$$

where $X = [x_1, x_2, ..., x_N] \in \mathbb{R}^N$, and $Softmax(X)_i$ is the i^{th} element in the softmax vector. Implementing a Softmax layer in hardware is a challenging task due to the complexity of exponential units. Besides the large number of hardware resources that are required to perform the exponential operations, overflow is another issue that must be considered. To tackle the overflow challenge, a down-scaling technique was proposed in [14], in which the Softmax function is re-defined as:

$$Softmax(X)_i = \frac{e^{x_i}}{\sum_{j=1}^{N} e^{x_j}} = \frac{e^{x_i}}{\sum_{j=1}^{N} e^{x_j}} \times \frac{e^{-m}}{e^{-m}} = \frac{e^{x_i - m}}{\sum_{j=1}^{N} e^{x_j - m}} \tag{8}$$

where $m = max([x_1, x_2, ..., x_N])$. This approach limits the outputs of the exponential units to the range of [0, 1].

Fig. 8 shows our proposed architecture for the Softmax layer. We keep the input values of a Softmax layer as Fixed<1,32,12>, and for the outliers that cannot fit in this format, we can clip the number such that it remains in the allowable range. Fixed<1,32,12> represents signed fixed-point numbers with 1 bit set aside for "sign", a total of 32 bits, out of which 12 bits are for the fractional part of the number, and the rest of the 19 bits for the integer part. Next, we subtract the maximum input value from the input vector to prevent the exponential operations from overflowing. If x < -8, the exponential function can be approximated with zero. On the other hand, $\forall x \in [-8, 0] : exp(x) \in (0, 1]$, that means we can keep the input and output value of the exponential units as Fixed<1,16,12> and Fixed<0,16,15>, respectively. As a result, we can use our method at a 16-bit resolution to calculate exponential computations efficiently.

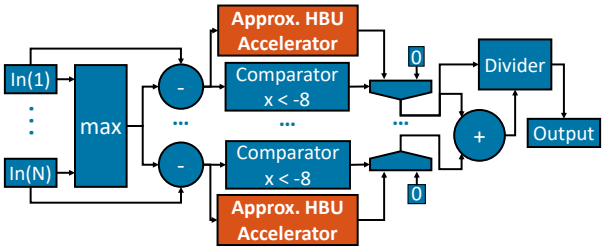


Figure 8: The proposed architecture for the Softmax layer.

Afterward, we use a 32-bit binary accumulator to add up all the exponential values from the HBU units. Finally, we use a binary divider to normalize the final output.

GELU (Gaussian Error Linear Unit) is another neural network activation function [13] that can be used in Transformers. Due to the non-monotonicity of this activation, GELUs can enable a neural network to model more complicated problems.

We keep the input of a GELU layer as Fixed<1,32,8>. In our method, we split the function into three separate sub-functions, two of which are linearly approximated with RELU. We define "Trimmed GELU" as follows.

$$TGELU(x) = \begin{cases} 0 & x < -8 \\ x & x \geqslant +8 \\ GELU(x) & O.W. \end{cases} \quad (9)$$

This function near zero ($x \in [-8, 8)$), however, has a complex behavior that must be implemented with high precision in Transformers, otherwise, it results in a significant accuracy degradation [16]. Since $\forall x \in [-8, 8) : GELU(x) \in (-1, 8)$, the input and output of the function in this region can be represented as Fixed<1,16,12>, because only 3 bits are needed to represent the integer part with a maximum magnitude of 8, and 12 fractional bits are enough for the accuracy we need. We deploy our method to perform the computation of this part. Fig. 9 shows our proposed architecture for the TGELU layer.

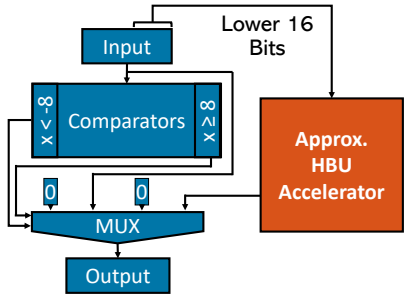


Figure 9: The proposed architecture for the GELU layer.

The accuracy evaluation was carried out on several GLUE development sets [34], each of which relates to a specific NLP task. We modified the GELU and Softmax layers in the Python implementation of I-BERT using the RoBERTa-based model [18] and the Fairseq toolkit [24]. The Python implementation is faithful to the number representation (floating-point for RoBERTa, fixed-point for the rest with the bit widths listed before), and the approximation method used. Table 4 reports the accuracy results.

Table 4: Accuracy comparison of BERT models.

Method	MRPC	CoLA	RTE	QNLI
RoBERTa [18]	89	84.4	79.4	92.7
I-BERT [16]	90	82.6	77.6	92.5
Exact HBU [6]	89.7	82.8	76.9	92.5
Our method (*Config. 1*)	87.5	80.2	67.5	86.7
Our method (*Config. 2*)	89	82.8	78.7	92.8

For evaluating our proposed architectures in terms of hardware resource utilization, we implemented an encoder block of BERT on a Xilinx XC7K70TFBG676-2 FPGA using Vitis-HLS 2020.2 (Fig. 7). Due to the design of BERT, one encoder block is architecturally similar to other encoder blocks, hence an accelerator for one block can be used for all the remaining blocks. Therefore, the accelerator design in Fig. 7 can support all operations in an encoder block by doing multiple passes through it. For example, we can break a matrix-multiplication of size 3072 into 24 passes through an accelerator which uses matrix-multiply and accumulate on a vector length of 128.

Tables 5 and 6 report the hardware cost using the different methods. To put our hardware implementation of Softmax and GELU in the context of a transformer encoder block accelerator, we used hls4ml [4, 32] to implement a Matrix-Multiplication and Addition (MMA) block. The MMA block is implemented using the conventional binary method, where inputs and weights are 8-bit integers, while the accumulation and bias are 32-bit. We picked a vector size of 128, with a reuse factor of 64 to fit our FPGA device target. GELU block also works on a vector size of 128, while Softmax is limited to a vector size of 80. This is because the BERT encoder design only requires Softmax of vector length 80. LayerNorm was not implemented currently, due to a lack of support for the layer in hls4ml. Even though this is not a complete implementation, it is still valuable, as it shows the context in which we can evaluate potential hardware benefits of our approach: it shows the relative hardware sizes and the latency of the MMA, GELU, and Softmax blocks.

With these blocks, we can compose five flavors of BERT: each flavor would use the specific method listed in Table 5 for Softmax and GELU layers, but all five use the same MMA block. We can see that I-BERT implementations can explode in FF and LUT usage specifically due to the INT32-based computation that includes division. In IBERT, the exponential function is approximated by factoring the input and then applying a polynomial approximation. Our implementation bypasses both the factoring and polynomial approximation, by directly approximating the original function in fixed-point. While this reduces the bit-length representation, our model with approximated GELU and Softmax has comparable accuracy to IBERT in GLUE benchmarks (Table 4). The number of cycles used by the MMA layer is 67, which makes it the bottleneck in the exact HBU, *Config. 1* and *Config. 2* (those only need between 3 and 9 cycles to calculate Softmax and GELU). However, in the I-BERT implementation, GELU becomes the bottleneck with 111 cycles, and in RoBERTa, Softmax is the bottleneck with 543 cycles.

Table 5: FPGA hardware costs of the Softmax, GELU, and MMA layers.

Method	LUT	FF	DSP	BRAM	Cycles
Softmax Layer (vector size 80)					
RoBERTa [18]	129061	74581	949	0	543
I-BERT [16]	492324	610550	0	0	80
Exact HBU [6]	14596	6715	0	0	9
Our method (*Config. 1*)	6743	1884	0	0	3
Our method (*Config. 2*)	7623	1964	0	0	3
GELU Layer (vector size 128)					
RoBERTa [18]	696967	578696	3200	960	101
I-BERT [16]	13555	4512	0	0	111
Exact HBU [6]	11448	4569	0	0	4
Our method (*Config. 1*)	6369	4758	0	0	4
Our method (*Config. 2*)	7908	4758	0	0	4
MMA Layer (vector size 128)					
hls4ml [4, 32] (reuse factor 64)	33944	11830	128	171	65
hls4ml [4, 32] (reuse factor 4)	340425	13104	2048	1025	6

Table 6: Latency matched hardware combination.

Method	LUT	FF	DSP	BRAM	Cycles
I-BERT [16] (*Design 1*)	539823	626892	128	171	256
I-BERT [16] (*Design 2*)	846304	628166	2048	1025	111
Exact HBU [6]	366469	24388	2048	1025	9
Our method (*Config. 1*)	353537	19746	2048	1025	6
Our method (*Config. 2*)	355956	19826	2048	1025	6

4.4 Application: Roberts Cross Edge Detection

As a real image processing and computer vision application, we deployed our method to implement an 8-bit Roberts Cross edge detection algorithm, which performs square and square root functions as integral parts of the algorithm. Again, we synthesized two hardware designs with different mean absolute error ranges to evaluate the effect of approximation on such a system-level implementation. We also compared the system quality and cost of hardware generated by conventional binary, stochastic, exact HBU, and our proposed methods. For binary implementation, we used a fully pipelined Xilinx CORDIC (v6.0) LogiCORE to perform the square root function. Fig. 10 shows the output images using different methods. Table 7 provides the comparison of these methods in terms of hardware costs as well as the peak signal-to-noise ratio (PSNR) as an image quality metric. The reference output image was generated by Matlab using floating-point double precision.

As the results show, the hardware cost of the first configuration of our method (*Config. 1*) in terms of area × latency is 40.183%, 0.907%, and 0.015% of the exact HBU, conventional binary, and stochastic methods, respectively. Using the second configuration (*Config. 2*), the area × latency cost of our method becomes 78.504%, 1.77%, and 0.029% of the exact HBU, conventional binary, and stochastic methods, respectively. Concerning image quality, the second configuration of our method has higher PSNR than the stochastic approach, which translates to higher quality. The quality of this implementation is also close to the results of the exact methods. The first configuration, however, has a slightly lower PSNR than the stochastic method, although the area × latency of this configuration is 4 orders of magnitude better.

Table 7: FPGA hardware costs and image quality results of the Roberts Cross edge detection algorithm.

Method	Area (LUT)	T_{crit} (ns)	Cy	A×L	PSNR
Conventional Binary	209	6.72	8	11235.84	30.80
Conventional Stochastic [27]	303	4.49	512	696560.64	28.18
Exact HBU [6]	107	2.37	1	253.59	30.80
Our Method (*Config. 1*)	43	2.37	1	101.91	26.77
Our Method (*Config. 2*)	84	2.37	1	199.08	30.70

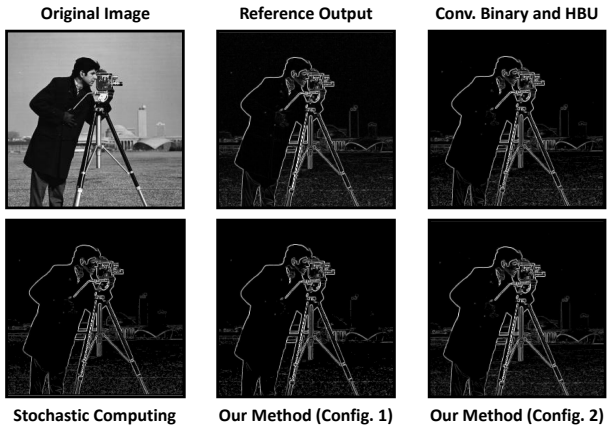


Figure 10: Roberts Cross edge detection algorithm's results.

5 CONCLUSIONS

We proposed a novel approach to implement almost all types of univariate math functions using ultra-low-cost hardware and yet preserving high accuracy. Based on the hardware implementations of some complex functions at 8- to 16-bit resolutions, we showed that our method outperforms the conventional binary, previous stochastic computing, and exact hybrid binary-unary methods in terms of hardware area, critical path delay, and latency. As for accuracy, our method on average introduces less error than conventional stochastic computing as an approximate approach. Moreover, our method can efficiently implement math functions at higher resolutions than the exact hybrid binary-unary work. By deploying our method to the non-linear operations in the encoders of the BERT language model and Roberts Cross edge detection algorithm, we also showed that our method results in much better area × latency than the other methods while achieving high-quality outputs.

ACKNOWLEDGEMENT

This material is based upon work supported in part by the National Science Foundation under grant number PFI-TT 2016390, and by Cisco Systems, Inc. under grant number 1044678.

REFERENCES

[1] A. Alaghi, W. Qian, and J. P. Hayes. 2017. The Promise and Challenge of Stochastic Computing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* PP, 99 (2017), 1–1.

[2] Florent De Dinechin and Bogdan Pasca. 2011. Designing custom arithmetic data paths with FloPoCo. *IEEE Design & Test of Computers* 28, 4 (2011), 18–27.

[3] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[4] J. Duarte, S. Han, P. Harris, S. Jindariani, E. Kreinar, B. Kreis, J. Ngadiuba, M. Pierini, R. Rivera, N. Tran, and Z. Wu. 2018. Fast inference of deep neural networks in FPGAs for particle physics. *Journal of Instrumentation* 13, 07 (jul 2018), P07027–P07027. https://doi.org/10.1088/1748-0221/13/07/p07027

[5] S. Rasoul. Faraji, Pierre Abillama, Gaurav Singh, and Kia Bazargan. 2020. HBUC-NNA: Hybrid Binary-Unary Convolutional Neural Network Accelerator. In *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*. https://doi.org/ISCAS.2020

[6] S Rasoul Faraji and Kia Bazargan. 2020. Hybrid binary-unary hardware accelerator. *IEEE Trans. Comput.* 69, 9 (2020), 1308–1319.

[7] S Rasoul Faraji and Kia Bazargan. 2020. Hybrid binary-unary truncated multiplication for DSP Applications on FPGAs. In *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. IEEE, 1–9.

[8] S. Rasoul Faraji, Gaurav Singh, and Kia Bazargan. 2019. HBUNN - Hybrid Binary-Unary Neural Network: Realizing a Complete CNN on an FPGA. In *IEEE International Conference on Computer Design (ICCD) (ICCD '19)*.

[9] N. Eamon Gaffney and Armin Alaghi. 2016. *scsynth*. https://github.com/arminalaghi/scsynth

[10] B.R. Gaines. 1969. Stochastic Computing Systems. In *Advances in Information Systems Science*. Springer US, 37–172. http://dx.doi.org/10.1007/978-1-4899-5841-9_2

[11] Jonas Gehring, Michael Auli, David Grangier, Denis Yarats, and Yann N Dauphin. 2017. Convolutional sequence to sequence learning. In *International conference on machine learning*. PMLR, 1243–1252.

[12] Alex Graves. 2012. Sequence transduction with recurrent neural networks. *arXiv preprint arXiv:1211.3711* (2012).

[13] Dan Hendrycks and Kevin Gimpel. 2016. Gaussian error linear units (gelus). *arXiv preprint arXiv:1606.08415* (2016).

[14] Ruofei Hu, Binren Tian, Shouyi Yin, and Shaojun Wei. 2018. Efficient hardware architecture of softmax layer in deep neural network. In *2018 IEEE 23rd International Conference on Digital Signal Processing (DSP)*. IEEE, 1–5.

[15] Devon Jenson and Marc Riedel. 2016. A Deterministic Approach to Stochastic Computation. In *Proceedings of the 35th International Conference on Computer-Aided Design* (Austin, Texas) *(ICCAD '16)*. New York, NY, USA, Article 102, 8 pages. https://doi.org/10.1145/2966986.2966988

[16] Sehoon Kim, Amir Gholami, Zhewei Yao, Michael W Mahoney, and Kurt Keutzer. 2021. I-bert: Integer-only bert quantization. In *International conference on machine learning*. PMLR, 5506–5518.

[17] Peng Li, D.J. Lilja, W. Qian, M.D. Riedel, and K. Bazargan. 2014. Logical Computation on Stochastic Bit Streams with Linear Finite-State Machines. *Computers, IEEE Transactions on* 63, 6 (June 2014), 1474–1486. https://doi.org/10.1109/TC.2012.231

[18] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692* (2019).

[19] Soheil Mohajer, Zhiheng Wang, and Kia Bazargan. 2018. Routing Magic: Performing Computations Using Routing Networks and Voting Logic on Unary Encoded Data. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Monterey, CALIFORNIA, USA) *(FPGA '18)*. ACM, New York, NY, USA, 77–86.

[20] Soheil Mohajer, Zhiheng Wang, Kia Bazargan, and Yuyang Li. 2020. Parallel unary computing based on function derivatives. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 14, 1 (2020), 1–25.

[21] M. H. Najafi, S. R. Faraji, B. Li, D. J. Lilja, and K. Bazargan. 2019. Accelerating Deterministic Bit-Stream Computing with Resolution Splitting. In *20th International Symposium on Quality Electronic Design (ISQED)*. 157–162. https://doi.org/10.1109/ISQED.2019.8697443

[22] M. Hassan Najafi, David J. Lilja, and Marc Riedel. 2018. Deterministic Methods for Stochastic Computing Using Low-discrepancy Sequences. In *Proceedings of the International Conference on Computer-Aided Design* (San Diego, California) *(ICCAD '18)*. ACM, New York, NY, USA, Article 51, 8 pages. https://doi.org/10.1145/3240765.3240797

[23] M. Hassan Najafi, D. J. Lilja, M. D. Riedel, and K. Bazargan. 2018. Low-Cost Sorting Network Circuits Using Unary Processing. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 26, 8 (Aug 2018), 1471–1480.

[24] Myle Ott, Sergey Edunov, Alexei Baevski, Angela Fan, Sam Gross, Nathan Ng, David Grangier, and Michael Auli. 2019. fairseq: A fast, extensible toolkit for sequence modeling. *arXiv preprint arXiv:1904.01038* (2019).

[25] W.J. Poppelbaum, A. Dollas, J.B. Glickman, and C. O'Toole. 1987. Unary Processing. In *Advances in Computers*. Vol. 26. Elsevier, 47 – 92.

[26] W. J. Poppelbaum, C. Afuso, and J. W. Esch. 1967. Stochastic Computing Elements and Systems. In *Proceedings of the Joint Computer Conference* (Anaheim, California) *(AFIPS '67 (Fall))*. ACM, New York, NY, USA, 635–644. https://doi.org/10.1145/1465611.1465696

[27] Weikang Qian, Xin Li, Marc D. Riedel, Kia Bazargan, and David J. Lilja. 2011. An Architecture for Fault-Tolerant Computation with Stochastic Logic. *IEEE Trans. Comput.* 60, 1 (2011), 93–105. https://doi.org/10.1109/TC.2010.202

[28] W. Qian and M.D. Riedel. 2008. The Synthesis of Robust Polynomial Arithmetic with Stochastic Logic. In *45th ACM/IEEE Design Automation Conference, DAC'08*. 648–653.

[29] Weikang Qian, Marc D. Riedel, and Ivo Rosenberg. 2011. Uniform Approximation and Bernstein Polynomials with Coefficients in the Unit Interval. *Eur. J. Comb.* 32, 3 (April 2011), 448–463. https://doi.org/10.1016/j.ejc.2010.11.004

[30] Sayed Ahmad Salehi, Yin Liu, Marc D. Riedel, and Keshab K. Parhi. 2017. Computing Polynomials with Positive Coefficients Using Stochastic Logic by Double-NAND Expansion. In *Proceedings of the on Great Lakes Symposium on VLSI 2017* (Banff, Alberta, Canada) *(GLSVLSI '17)*. ACM, New York, NY, USA, 471–474. https://doi.org/10.1145/3060403.3060410

[31] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[32] vloncar, Sioni Summers, Javier Duarte, Nhan Tran, Ben Kreis, jngadiub, Nicolò Ghielmetti, Duc Hoang, EJ Kreinar, Kelvin Lin, Maksymilian Graczyk, Adrian Alan Pol, ngpaladi, Dejan Golubovic, Yutaro Iiyama, Zhenbin Wu, Delon, Paolo Cretaro, veyron8800, Anders Wind, David, GDG, Jovan Mitrevski, Konstantin Vinogradov, Konstantin Vinogradov, Petr Zejdl, Sarun Nuntaviriyakul, Thea Aarrestad, and drankincms. 2021. *fastmachinelearning/hls4ml: coris*. https://doi.org/10.5281/zenodo.5680908

[33] John Von Neumann. 1956. Probabilistic logics and the synthesis of reliable organisms from unreliable components. *Automata studies* 34, 34 (1956), 43–98.

[34] Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R Bowman. 2018. GLUE: A multi-task benchmark and analysis platform for natural language understanding. *arXiv preprint arXiv:1804.07461* (2018).

[35] Zhiheng Wang, Naman Saraf, Kia Bazargan, and Arnd Scheel. 2015. Randomness Meets Feedback: Stochastic Implementation of Logistic Map Dynamical System. In *Proceedings of the 52Nd Annual Design Automation Conference* (San Francisco, California) *(DAC '15)*. ACM, New York, NY, USA, Article 132, 7 pages. https://doi.org/10.1145/2744769.2744898